

Integrating CBSE, SoC, MDA, and AOP in a Software Development Method

Raul Silaghi and Alfred Strohmeier
Software Engineering Laboratory
Swiss Federal Institute of Technology in Lausanne
CH-1015 Lausanne EPFL, Switzerland
{Raul.Silaghi, Alfred.Strohmeier}@epfl.ch

Abstract

Component-Based Software Engineering, Separation of Concerns, Model-Driven Architecture, and Aspect-Oriented Programming are four active research areas that have been around for several years now. In this paper, we present how these four paradigms can be put together in the context of a new software development method and we show how they can complement each other at different stages in the development life-cycle of enterprise, middleware-mediated applications. Different software development methods, such as Fondue, Catalysis, Kobra, and the Rational Unified Process, are also analyzed, pointing out their differences and limitations. In the end, requirements for a dedicated tool infrastructure that would support the new development approach are discussed.

1. Introduction

Many people have been working over the last few years on the four paradigms that are mentioned in the title, building research communities that exist on their own. The four big words in the title should not put the reader off, and the paper should be seen more like a vision towards a future method that would bring together the four communities, will combine their strengths, and will complement each other to overcome their weaknesses.

A software development method consists of a set of concepts, a defined notation, a specified process, and a collection of heuristics. The concepts are the building blocks of description. They capture what it is possible to express, for instance, a system may be comprised of a hierarchy of classes related through inheritance, or a system may consist of a hierarchy of inter-connected components. The notation is the syntax that expresses the concepts. The process describes the sequence in which the pieces of notation are constructed or presented. Finally the heuristics capture the informal and pragmatic guidelines which allow the developer

to construct and evaluate the various diagrams prescribed by the process.

Building distributed enterprise applications that require the interoperation of multiple components that may be distributed, independently operated, and heterogeneous with respect to language, data model, environment, architecture, and protocols, is a non-trivial task. A middleware is required in order to integrate these diverse software components and to allow them to interoperate effectively. In order to ease the job of software developers and to guide them through the development life-cycle of such enterprise, middleware-mediated systems, a new software development method, called *Enterprise Fondue*, will be proposed in this paper.

Component-Based Software Engineering (CBSE) is an evolutionary rather than revolutionary approach. Its concepts and ideas stem from the long-known principles of encapsulation and modularization, from the “divide-and-conquer” approach, and from modular programming and object-orientation. During the last few years, due to the rapid development of Internet technology and of enterprise applications, CBSE was seen to be the best strategy for on-time and high-quality solutions. By using the CBSE approach, system development becomes the selection, reconfiguration, adaptation, assembling and deployment of encapsulated, replaceable and reusable system elements called *components*, rather than building the system from scratch [1]. So far, the component paradigm has been introduced mainly through the new technological solutions and distributed component infrastructures, such as Microsoft’s (Distributed) Component Object Model (COM, DCOM) [2], Object Management Group’s Common Object Request Broker Architecture (CORBA) [3], or Sun’s Enterprise JavaBeans (EJB) [4]. The “component vs. object” issue has become the focus of many discussions and studies, raising the question whether and/or to what extent object-orientation can provide effective support for component-based system development [1], [5].

Separation of concerns is an approach to decomposing software into smaller, more manageable and comprehensible parts, each of which deals with, and encapsulates, a particular area of interest, called a *concern*. Basically, a concern can be viewed as anything that is of importance to the application, be it infrastructure, code, requirements, design artifacts, etc. Separation of concerns provides support to overcome the “tyranny of the dominant decomposition” [6], from which many modern artifact notations suffer. For example, object-oriented approaches provide mechanisms to encapsulate certain kinds of concerns, such as data and functions. However, they do not provide mechanisms that would allow one to encapsulate cross-cutting concerns, such as distribution or security, in an effective way. To help solve the various problems related to poor separation of concerns, several advanced modularization mechanisms have been developed over the last decade. These include role-modeling [7], subject-oriented programming [8], viewpoints [9], adaptive programming [10], aspect-oriented programming [11], adaptive plug-and-play components [12], multi-dimensional separation of concerns and hyperspaces [6]. Unfortunately, despite its importance throughout the software life-cycle, a large amount of work on advanced separation of concerns has targeted separation of concerns in code artifacts. Hyper/J [13] and AspectJ [14] are two well-known tools that support Advanced Separation of Concerns at the code level.

Model Driven Architecture (MDA) is OMG’s new approach to software architecture that provides a standardized way for specifying information systems by clearly separating the “what” and the “how”, or as said in [15] “... that separates the specification of system functionality from the specification of the implementation of that functionality on a specific technology platform”. Both specifications are expressed as models: *Platform Independent Models (PIMs)*, which specify the structure and functions of a system while abstracting away technical details, and *Platform Specific Models (PSMs)*, which are derived from PIMs and specify how the functionality is to be realized on a selected platform. Moving from one model to another is achieved by applying model transformations, and finally, code generators are used to map the most specialized PSM to a specific technology platform, such as CORBA, J2EE, .NET, or Web Services. More details and documents on MDA can be found in [16].

Aspect-Oriented Programming (AOP) [11] has been proposed as a technique for improving separation of concerns in software. This approach makes it possible to separately specify various kinds of concerns and localize them into separate units of encapsulation, called *aspects*. One can deal with both the concerns and the modules that encapsulate them at different levels of abstraction, not only at the code level. AspectJ [14] is a general-purpose aspect-orient-

ed extension to Java that introduces concepts like *join points* and *pointcuts* to describe the structure of the cross-cutting concerns, and *advices* to specify the desired behavior to be performed throughout the identified structure. Similar extensions exist to other programming languages, such as C++, C, Smalltalk, or Ruby.

When looking at these four paradigms they do not seem to have much in common. However, they can complement each other at different stages in the development life-cycle of enterprise applications as we will show in this paper. A new software development method, called *Enterprise Fondue*, that makes these emerging technologies work together will be presented. Starting from a component-based description of the system to be implemented, the developers continue by designing the inner structure of all the identified components. The obtained models are further refined along several concern-dimensions. During the last phase, code generators are used to map the different models to code, and aspects are used to weave-in code corresponding to the previously considered concerns.

The rest of the paper is organized as follows: Section 2 describes some of the currently existing object- and component-oriented software development methods; Section 3 motivates the continuous need for modeling enterprise, middleware-mediated applications in the presence of Web Services; Section 4 introduces our new software development method *Enterprise Fondue* based on CBSE, SoC, MDA, and AOP; different layers are identified and their intended content and role is explained, showing also how to move from one layer to the next; Section 5 proposes some requirements that tool vendors should meet if they decide to support the new development method, and Section 6 draws some conclusions.

2. Software Development Methods

There is a large number of different methods to choose from for any new software development project, with an equally large number of different processes and modeling approaches. Major methods, such as Fondue, Catalysis, Kobra, and RUP, all claim to provide a holistic approach to software development, supporting the seamless development of software systems from early analysis to executable code. These methods are briefly described in this section, pointing out their (common) origins, their differences, and their limitations. Readers that are familiar with these methods may skip this section and jump directly to section 3.

2.1. Fondue

Fondue [17], [18] is an object-oriented software development method that is based on the original Fusion method [19]. It not only provides the internal view of the class mod-

el and the behavior of individual classes, but it includes modeling of system-wide functionality and a step-by-step process that leads the development team from an initial requirements document through to the implementation of an object-oriented software system. Fondue keeps the process and the models of the original Fusion method but uses the Unified Modeling Language (UML) [20] as a notation. In addition to the graphical notation of UML, Fondue specifies operations by Operation Schemas, which describe the effect of the operations on an abstract state representation of the system and the messages sent to the outside world. Operation Schemas describe the assumed initial state by a precondition, and the required change in system state after the execution of the operation by a postcondition, which are both written in UML's Object Constraint Language (OCL) [21]. In addition to Fusion, Fondue makes use of use cases during requirements elicitation. Instead of regular expressions as in classical Fusion, a restricted form of state diagrams is used for describing sequencing of system operations. To the contrary of classical Fusion, the Concept Model (Fondue's equivalent to Fusion's object model) is refined into a Design Class Model, and finally an Implementation Class Model. The Fondue process can be outlined by a UML class diagram showing usage dependencies between models. It is provided in two parts, the Fondue Analysis Process and Models, and the Fondue Design Process and Models [18].

The current version of Fondue does not fully address concerns that are related to distributed systems, and especially middleware-specific concerns. A detailed study of the Fondue models is currently undergoing for deciding which are the most appropriate places to integrate different middleware-related concerns, such as distribution, transactions, concurrency, or security.

2.2. Catalysis

The development of the Catalysis method [5] started in 1991 as a formalization of the Object Modeling Technique (OMT) [22] but soon also became an extension of recent OMT variants, such as Fusion [19]. Catalysis is aimed at providing a unified, component-based development process, by combining the strengths of the 'early' methods in analysis and design with a systematic treatment of refinement and architectural design.

Although it is based on a small number of underlying concepts, namely type, collaboration, refinement and framework, Catalysis leaves the impression of a complex method. As a matter of fact, it uses an iterative and incremental process based on clearly defined abstraction and refinement mechanisms. System development is viewed as a series of refinements, in which translation is regarded as a special form of refinement. These refinements are applied throughout the whole life-cycle, from early analysis to im-

plementation. Each refinement step is designed to systematically lower the level of abstraction towards code in a high-level language. However, Catalysis does not define an explicit barrier for the refinement process. Besides mentioning that the refinement process stops at a level of abstraction close enough to code, which can then be refined (i.e., mapped, translated) to code, Catalysis does not define until what level the refinement process should proceed to describe all 'major' decisions. Moreover, it does not address the impact that non-functional requirements may have. This may result in abstract models that are not yet implementable or models that violate the system's quality requirements.

2.3. RUP

Coming later in the evolution of object-oriented methods, the Rational Unified Process (RUP) [23] is a specific and detailed instance of a more generic process, the Unified Process (UP) [24], but more sophisticated than either OMT or Fusion. It actually represents an amalgam of OMT [22], Objectory [25], Booch [26], and the "Rational Approach" [27]. The RUP has been developed to provide a unified process to support the full power of UML. According to [24], the process may be characterized as a component-based, use-case-driven, architecture-centric, iterative, and incremental software development method. In principle, the RUP iterates over a series of cycles where a cycle consists of four phases: Inception, Elaboration, Construction, and Transition. In addition, the process defines various workflows, the most prominent being Requirements, Analysis, Design, Implementation, and Test, which are carried out to a specific extent in each phase of a cycle. These workflows are very similar to the four activities, i.e., coding, testing, listening, and designing, that Beck places at the basis of eXtreme Programming and of its twelve practices [28]. Other RUP workflows are related to management, such as project management, configuration and change management, or to process configuration, such as the environment workflow.

The RUP uses the results of the design workflow to implement the design classes in terms of components, which are considered to be "physical packages of programming code". During design, many details of a class and its relationships are described using the syntax of the chosen programming language which makes code generation straightforward. In particular, this is enforced for operations and attributes of a class, as well as for the relationships in which the class participates [24]. However, this bears the danger of misusing the UML as a graphical programming language, and of invalidating models by changing the programming language. In addition, the RUP requires the 'refinement' of abstract constructs to constructs of the chosen programming language, but provides only abstract suggestions for doing so. This is even true for the description of class relation-

ships. Furthermore, the impact of non-functional requirements is considered only within the Requirements workflow, neglecting the impact of such requirements in the modeling and implementation of a system.

2.4. Kobra

The Kobra method, first introduced in [29] and then fully described in [30], has been influenced by, and has similarities with other software development methods, particularly Cleanroom [31], Fusion [19], and Catalysis [5]. It is also compatible with RUP [23] and OPEN [32]. The aim of Kobra is to provide concrete support for the development and application of component-based, domain-specific frameworks.

The central artifact of the Kobra method is the framework, i.e., a generic description of a family of applications, which encapsulates not only the common parts but all concrete variants as well. This is achieved by capturing all possible features within the framework and using decision models to describe the choices that distinguish distinct members of the family. To develop a concrete application, the generic framework is instantiated by resolving all decision models. In principle, an application removes the genericity within a framework, but does not change the level of abstraction at which it is described. Frameworks as well as applications are described by UML diagrams at a level of abstraction similar to that used to describe a design. Therefore, it is necessary to transform these models into a form that can be understood by compilers. Among other things, this includes source-code in a particular language. However, multiple implementations can be created from a given application. Each implementation can then be used to create executable images of a system.

The Kobra method makes use of a technique, known as SORT (Systematic Object-oriented Refinement and Translation), for implementing object-oriented models based on the principle of distinguishing and strictly separating between refinement and translation activities. To this end, SORT makes use of pattern technology to support such activities. The SORT technique is discussed in more detail in [33] and [34].

3. Enterprise Software Development – Vision

In this section, we present the advantages and disadvantages of Web Services with respect to currently existing middleware infrastructures. We also explain how these infrastructures actually support the implementation of Web Services, and how the two actually act at two different levels of abstraction.

The arrival of Web Services is viewed by many as the dawn of a new area of interoperability as it promises to link

disparate businesses in a manner and scale reminiscent of the way the Internet, with TCP/IP, linked machines. Web Services will allow businesses to talk to each other, free of the shackles of platform and protocol. Multiple departments, both within and outside of an organization, will reuse application functionality that is decomposed and offered as a Web Service. A new breed of applications will be “composed” and “integrated”, rather than “built”, and “pay-per-use” business models will become popular.

But, haven’t we heard this before? Isn’t interoperability, platform independence, and the reuse of distributed components exactly what CORBA promised? The straight answer would be yes. However, Web Services address these problems in a somehow different way making them more eligible to success in a time when the emphasis on interoperability is much greater than it was with CORBA.

For a start, in most cases, administrators will simply not open firewalls to let protocols through, such as the Internet Inter-ORB Protocol (IIOP), used by CORBA, or the Distributed Computing Environment Remote Procedure Call (DCE RPC), used by DCOM. In the absence of a vendor-independent and popular means to traverse firewalls, CORBA has often been confined within the corporate firewall. Since

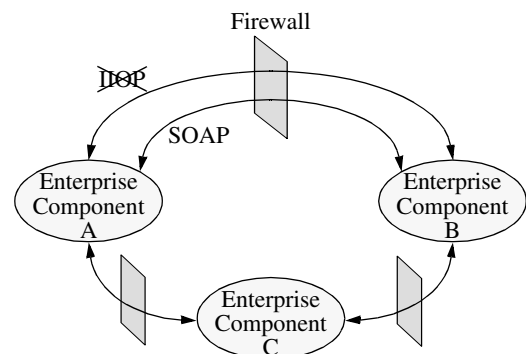


Figure 1. Firewalls between enterprises

the Simple Object Access Protocol (SOAP), which is the wire protocol for Web Services, can be carried over firewall-friendly HTTP (the network protocol), it can sail through corporate firewalls.

In addition, SOAP carries data encoded in the eXtensible Markup Language (XML), which is text and not binary, as in the case of the Common Data Representation (CDR), used by IIOP, or the Network Data Representation (NDR), used by DCE RPC. Even though CORBA’s wire encoding is highly efficient, it is prone to interoperability problems by virtue of being binary. Every single byte matters, as does the order.

Besides its asynchronous nature, what makes SOAP even more attractive is the fact that XML parsers and HTTP are low cost and ubiquitous. In fact, SOAP clients might become the clients of choice on wireless devices because it is

easier to have an XML parser on a wireless device than an ORB.

The road of Web Services is not without its bumps though. First of all, the specifications of SOAP, UDDI (Universal Description, Discovery and Integration), and WSDL (Web Services Description Language), upon which Web Services rely on, are continually evolving. While the mass movement towards Web Services by most software vendors brings great joy, the current reality of interoperability across vendors' products leaves much to be desired. Second of all, important concerns that come in mind when thinking of e-business across corporations are those of security and transactional integrity, which, unfortunately, have not yet been addressed by the Web Service specifications, and vendors offer only proprietary solutions.

Once all these problems will be solved, we believe that Web Services will become the standard for inter-enterprise communication, as shown in Figure 1. On the other hand, CORBA, which has matured into a technology with widely adhered-to standards regarding distributed object management, quality of service, and with popular services providing Naming, Transactions, Notification, Security, to name just a few, will remain the solution of choice when it comes to enterprise computing across languages and platforms. CORBA technology allows legacy applications to become Web-enabled or otherwise interoperate. We believe that CORBA will continue to be used, with the CORBA servers "exposed" as Web Services (for the Internet), protecting investments and allowing high-performance IIOP-based access within the enterprise (the intranet).

Extending the variety of technologies that can be used inside an enterprise, we believe that an enterprise component from Figure 1, can further be decomposed into "islands", as presented in Figure 2. In order to support this idea with an example, one could imagine that the different islands correspond to different departments inside the same enterprise, such as Accounting, Human Resources, Public Relations, and so on. Each department might have moved towards an IT solution (decentralized at the beginning) at different moments in time, thus considering the latest technology that was available at that time. Or, to give a more realistic example, these islands could very well be the result of several corporate mergers and acquisitions.

With this section we wanted to stress out that even though Web Services seem to be such a good solution for inter-enterprise communication (Business-to-Business), they still have to be implemented using one of the currently existing middleware infrastructures, and thus, there is still a need for modeling such enterprise, middleware-mediated systems before being able to expose the implemented functionality as a Web Service.

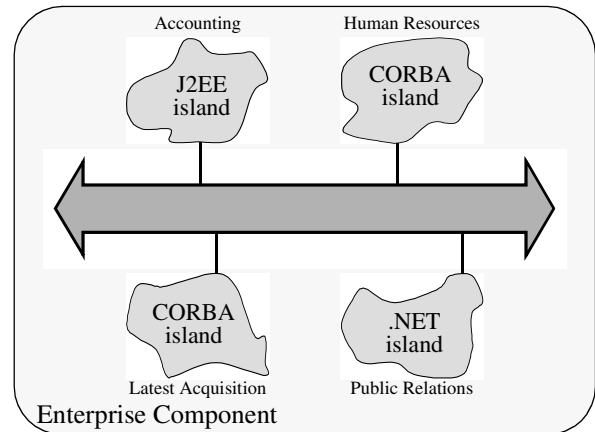


Figure 2. Inside view of an Enterprise Component

4. A New Software Development Method Based on CBSE, SoC, MDA, and AOP

In this section, we will present a new software development method, *Enterprise Fondue*, from its early life-cycle stages when developers and stakeholders define a first high-level architecture of the enterprise system, and down to the least implementation details that will actually build the system under consideration. As we will see, the *Enterprise Fondue* method capitalizes on the previously described methods by integrating some of their best features, and by trying to overcome some of the limitations we described in section 2, e.g., lack of distribution and explicit components in Fondue, no explicit barrier between models and code in Catalysis, and so on. Regarding the software process, *Enterprise Fondue* adopts primarily a top-down approach rather than an incremental and iterative one as does RUP. At the implementation level, *Enterprise Fondue* complements KorbA by using aspects, which are certainly a nice mechanism for encapsulating crosscutting concerns.

Figure 3 presents the different layers that we have identified, their contents and the way in which the enterprise system is supposed to evolve from one layer to the next.

At the highest-level of abstraction, the *Component-Based Layer*, the developers together with the stakeholders will describe the system to be implemented in terms of components and relationships between them. Certain functional and non-functional concerns will be identified and different functionalities will be assigned to different components. At the second layer, the *Concern-Driven Object-Oriented Models Layer*, the developers will start designing the internal structure of all the components by using models and by refining these models along different concern-dimensions. The third layer, the *Technology Dependent Layer*, is obtained by refining along the technology-dimension. Each such refinement results in a technology-specific view of our

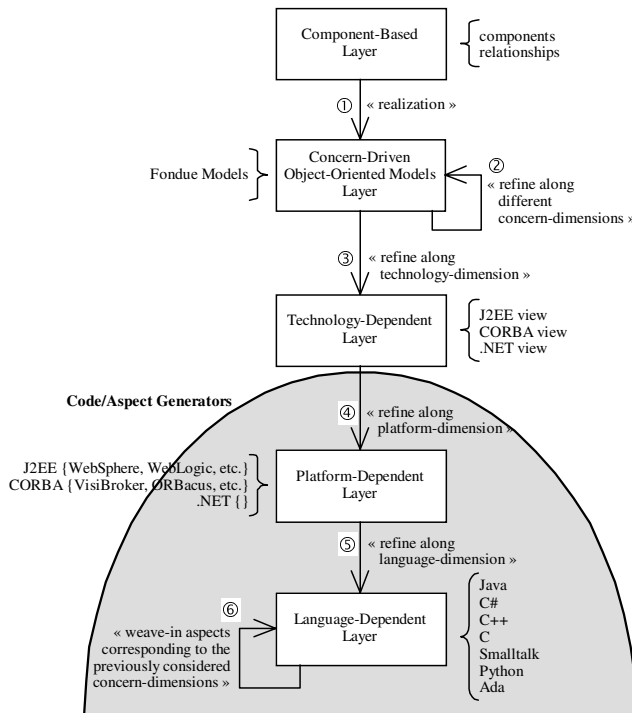


Figure 3. A new software development method

system, e.g., J2EE-view, CORBA-view, or .NET-view. After this layer, code generators are used to map the different technology-views to concrete middleware implementation platforms, such as VisiBroker, ORBacus, Orbix2000, or VoyagerORB for the CORBA-view, and WebLogic, WebSphere, or Orbix E2A for the J2EE-view. For some mappings, the implementation language is fixed in advance, e.g., the Java language will be used for mapping the J2EE-view to a particular J2EE platform. For others, different target implementation languages might be chosen depending on the ones that are actually supported. It might be the case that a particular CORBA platform has implemented only the IDL to C++ mapping, in which case the target implementation language cannot be other than C++. Even though the last two refinement steps are impossible to separate, they still address two different concerns, namely the platform (*Platform Dependent Layer*) and the implementation language (*Language Dependent Layer*). Once the object-oriented code has been generated, aspects are used to weave-in specific code that will actually implement the previously considered concerns.

The rest of the section describes in more details all the layers, the notation to be used at each layer, and how the refinement process will end up with code that implements the desired system.

4.1. The Component-Based Layer

One of the important characteristics and benefits of the CBSE approach is, in our opinion, the fact that the component concept represents an excellent solution for providing a meeting point between the different stakeholders and the developers. By defining a component as an encapsulated concept, clearly delimited from the environment, with specific roles and behavior in the domain, with hidden interior and exposed functionality through interfaces, it can be easily understood by both worlds. Such a concept gives business analysts and managers greater ability to model business processes and requirements at a higher-level, in a domain-specific, but implementation-independent way. On the other hand, the application developers retain control over how their models are turned into complete applications using advanced component-based technology infrastructures.

Object-orientation cannot be effectively used for that purpose. Objects can be too small in size and too technology-oriented to be considered as basic units of a development process by business-oriented people. The logic is usually too trivial to justify the expense of modeling, building, documenting, and reusing a single object interface. On the other hand, business processes are often too fuzzy and complex to be uniformly and easily understood by IT specialists. They must be broken down into constituent, semantically separated business building blocks in order to be efficiently handled by application developers. Furthermore, components, as behavior- or process-based concepts, handled through the services they offer, represent a more natural approach for describing complex business processes than objects as entity-based structures. Thus, a component can represent a *lingua franca* for the business and IT worlds [35].

As it can be seen in Figure 4, an enterprise component is built out of several business components that cooperate to deliver the cohesive set of functionality required by a specific business need. At its turn, an enterprise component can be a constituent business component of a larger enterprise system. For the time being, at this high-level of abstraction, both the stakeholders and the developers are only interested in identifying business components and relationships between them. Later on, when technology decisions will be made, several business components will be grouped into specific technology islands depending on the technology that will be used to implement them.

As an example, suppose that we are requested to build a complete information system for a bank. In this case, the enterprise component would be the bank itself, which might expose a Web Service interface to the outside world so that possible clients may access the services that the bank provides for them. The output of the Component-Based Layer will be a hierarchy of components and the relationships be-

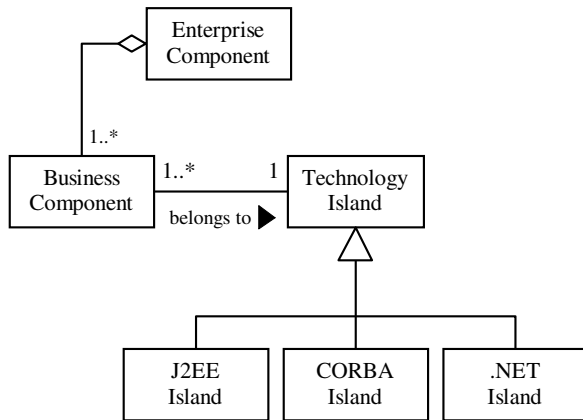


Figure 4. Enterprise Component structure

tween them. At the first layer we might find the different departments that make up the bank, such as Finance, Investment and Analysis, Trade and Export, Payment, Retirement and Insurance, etc., followed by finer grain components, such as Account, Client, Credit, Card, Fund, RealEstate, Product, PaymentSlip, MarketStock, etc.

Although the current version of UML does not provide all necessary elements and concepts for component-oriented modeling and specification, we believe that the available extension mechanisms, such as stereotypes, tagged values, and naming conventions, could very well be used for that purpose. Moreover, the UML Revision Task Force and the UML 2.0 Working Group are both giving high priority to component modeling related issues, promising a better support for components in the future UML.

4.2. Concern-Driven Object-Oriented Models Layer

After breaking the enterprise component into small business components in the previous layer, it is time now to start building those smaller components, i.e., we move down on the *<<realization>>* arrow in Figure 3 ①. The internal structure of a business component can be designed using any software development method that uses UML-compliant notations. Since the Fondue method was developed by members of our laboratory, we will consider this method for designing the identified business components.

As it was previously described, the Fondue method is an object-oriented method that requires several models to be built before obtaining a final description of the system, which is a business component in our case. The two major outcomes that a developer gets when applying the Fondue method are, in our opinion, the Design Class Model and the Interaction Model. The Design Class Model is represented using a UML class diagram and presents the decomposition of the system into classes, the internal structure of those

classes together with the functionality they provide, and different kinds of relationships (associations, aggregations, compositions, generalizations/specializations) between classes. The Interaction Model is a set of UML collaboration diagrams and presents how the different functionalities are actually implemented, how different classes interact in order to achieve a certain functionality.

Thus, by applying the Fondue method for each business component, we will obtain a set of class diagrams and collaboration diagrams, which all together represent the design of the different functionalities that were assigned to those components at the previous layer. However, besides the functionality concern, no other concerns have been addressed so far.

As already said at the beginning of this paper, middleware is an essential element in large distributed systems such as those that support enterprise applications, requiring multiple components to interoperate. Moreover, middleware, like software in general, is subject to concerns. Several concern-dimensions about middleware can be grouped into a category called Middleware Services, as the middleware addresses specific concerns of a system, such as communication, distribution, concurrency, security, or transactions. An extended list of categories that group several middleware-specific concern-dimensions can be found in [36]. In the rest of this paper, we will address how the middleware services concerns can be integrated in the already existing models (see Figure 3 ②) and how aspects can capture such concerns and weave them into the object-oriented code that only implements the pure functionality of the system under consideration (see Figure 3 ③). A similar approach should be applied when dealing with other concerns related to enterprise systems.

MDA identifies four types of model-to-model transformations (mappings) within the software development life-cycle [15]. The ones that we are proposing in Figure 3 ② are falling in the PIM-to-PIM type, i.e., they relate to platform-independent model refinement and are applied when PIMs are enhanced, filtered, or specialized. In our case, the enhancement or the specialization is performed along one concern-dimension, or one middleware-specific concern-dimension to be more precise. However, we cannot apply *one* predefined model transformation to refine all kinds of enterprise application models along *one* middleware-specific concern-dimension. The model transformation needs to be adapted to the application, otherwise it might refine the models in a wrong way. Since MDA does not have support for variability, *generic* model-to-model transformations have been proposed [37]. A generic transformation is specialized by using a set of parameters, which express the properties that are specific to a given application.

To conclude, in order to have model transformations that are concern-oriented and that can be specialized for particular application, we must have something like *generic concern-oriented model transformations* [38]. Such transformations will be first specialized according to the particularities of the current application. Once we have a specialized concern-oriented model transformation, it can be applied, and we will get refined models of the current application along the considered concern-dimension. In our method, several generic concern-oriented model transformations will be applied (see Figure 3 ②), refining the Fondue design models along the concern-dimensions that were identified at the previous layer.

One of the big problems at this layer is the absence of a clear notation for representing middleware-specific concerns, and concerns in general, inside UML models. Another problem is related to the composition of different concerns at the model level. While the first problem was and will be addressed by several workshops on “Aspect-Oriented Modeling with UML” ([39], [40]), the second one might be solved by proposing an exact workflow model that should specify the exact order in which concerns should be dealt with.

4.3. Technology Dependent Layer

Once a UML-compliant notation will be adopted for representing different middleware-specific concerns, the refinement along the technology-dimension (see Figure 3 ③) will be an easy and straight-forward step. The source model is represented using UML-compliant elements, some of which are the result of the Fondue design and some others are the result of the different refinements that were performed along different concern-dimensions. The target model is expressed using notations that are specific to the different UML profiles that currently exist. For instance, the refinement along the EJB-technology-dimension will result in the J2EE-view, which will be represented based on notations that can be found in the UML Profile for EJB [41]. The same applies for CORBA, in which case the CORBA-view will be represented using notations that can be found in the UML Profile for CORBA [42]. Since both the source and the target models use well defined notations, we believe that the refinement along the technology-dimension should be a completely automatic step achieved through technology-oriented model transformations. Moreover, since these transformations will be like one-to-one mappings, they will no longer depend on the application under development, and thus, genericity is no more required.

4.4. Platform Dependent Layer and Language Dependent Layer

The last two refinement steps, along the platform- and language-dimensions, are performed using code generators, which are considered to be a special kind of model transformations [15]. In addition, aspects will be used to weave-in code that addresses the different middleware-specific concerns that were considered in the second layer.

While there are quite a few code generators targeted towards widely used programming languages, such as Java or C++, not many are capable of generating platform specific code, i.e., code that addresses middleware-specific concerns, like distribution, transactions, or security.

Over the last few years, multiple attempts have tried to somehow *aspectize* away these middleware-specific concerns from the rest of the distributed application. Some of them have failed, some others have succeeded up to a certain level. However, as clearly described in [43], the big difficulty stems from the fact that, although the mechanisms used to implement these middleware-specific concerns are physically separated from the “functional” part of an application, they still remain semantically coupled. Thus, without having any idea about the application (semantics), it becomes impossible to apply, for example, a *general transactional aspect* to previously non-transactional code and to obtain the desired functionality, i.e., transactional behavior.

As already presented in [38], we believe that the application-specific set of parameters that is used to specialize the generic concern-oriented model transformations at layer two, could be further used to specialize aspects that will be weaved-in at layer five. A one-to-one association will exist between model transformations and aspects, i.e., a specialized concern-oriented model transformation will refine a model along one concern-dimension, and one specialized aspect will implement the concern at code level, respectively. In this way, since the specialized aspect has knowledge about the application, we might overcome the problem of semantic coupling described in [43]. Moreover, the order in which the specialized concern-oriented model transformations have been applied at model level will dictate the order in which aspects will be applied at code level, i.e., their precedence.

As a conclusion, instead of having one code generator which takes the most specialized model from layer three and generates platform specific code, we propose to rather have a code generator for the pure “functional” model of the application, and then have *aspect generators*, which generate specialized aspects from specialized concern-oriented model transformations, for the different middleware-specific concerns that the application needs to incorporate. The specialized aspects should be both platform- and language-

dependent, since, for example, it is not the same to weave-in a distribution aspect for ORBacus or for WebSphere.

5. Tool Support

Once we have a standardized UML-compliant notation for representing middleware-specific concerns (layers two and three) and know how to encapsulate them into aspects (layer five), it will be much easier for possible software vendors to provide a dedicated tool infrastructure that supports our new software development method *Enterprise Fondue*. Since our method is highly based on generic concern-oriented model transformations, it is our belief that the refinement process would be eased by providing facilities like:

- Support for generic model transformations as described in [37], together with support for testing pre- and post-conditions associated with model transformations.
- Concern-oriented wizards should be supported for configuring the generic model transformations along a concern-dimension.
- Plug-in support for code generators for different platforms and programming languages. They are supposed to map models to code.
- Plug-in support for aspect generators for different platforms and programming languages. They are supposed to map model transformations to aspects.
- Version management capabilities for the model repository. An Undo/Redo facility for model transformations would also be appreciated.
- Refactoring capabilities at model level (layers two and three in our method).
- Visual tools should be capable of demarcating model parts that have been added to the model through different specialized transformations by using different colors. In this way, we would be able to see what elements were introduced by which concern.
- Support for importing/exporting models in XMI format.
- Guidance in the refinement process. A workflow model could exist to track the refinement process through transformations. The workflow model could define which generic transformations can be applied at a certain refinement step, and therefore could determine the allowed sequence of transformations.

6. Conclusions

After a brief presentation of some of the currently existing object- and component-oriented software development methods, we argued on the continuous need for modeling enterprise, middleware-mediated applications, and we came up with a new development method, called *Enterprise*

Fondue, that is based on CBSE, SoC, MDA, and AOP. The new method identifies five different layers that correspond to different levels of abstraction in the development life-cycle of enterprise applications. At each layer, we clearly specified “what” needs to be addressed and “how” it can be represented. In fact, the notation to be used through the whole process is based on UML and its extension mechanisms.

The *component* concept was chosen to be used at the highest-level of abstraction since it represents an excellent solution for providing a meeting point between the two different worlds of developers and business stakeholders. Separation of concerns guides the refinement process, which is implemented using MDA model transformations. A new concept of *generic concern-oriented model transformation* has been proposed in order to support the variability that can appear from one application to another, on one hand, and the refinement along one concern-dimension, on the other hand. Finally, at implementation time, specialized *aspects* can be weaved into already existing object-oriented code in order to address the previously considered concerns.

The new method that we are proposing might look very similar to MDA. However, it is more than that since we are very specific about how the refinement process should progress. Moreover, we deal with generic transformations and, besides the mapping of models to code, we encourage the mapping of model transformations to aspects.

Although there is a lot of work to be done at all the layers in order to make the method more concrete, we believe that this paper has the merit to bring four important paradigms in software engineering together and to show how they can complement each other at different stages in the development life-cycle of enterprise, middleware-mediated applications. Now that we have the image of the big puzzle, we still need to find the pieces to solve it. Our next step will be to establish a well-defined set of UML-compliant notations for representing different middleware-specific concerns and to provide them (the notations) with a clear specification including graphical representation, meaning, behavior, interpretation, etc. Once we have a standardized notation, it will be much easier to start defining the concern-oriented model transformations.

The paper also discusses some challenging issues that tool vendors would need to address if they decide to support the new software development method.

References

- [1] Szyperski, C.: *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, 1998.
- [2] Microsoft, Inc.: *COM (Component Object Model), COM+, DCOM (Distributed COM)*. <http://www.microsoft.com/com/>.

- [3] Object Management Group, Inc.: *The Common Object Request Broker: Architecture and Specification*, v3.0, July 2002.
- [4] Sun Microsystems: *Enterprise JavaBeansTM Specification*, v2.0, August 2001.
- [5] D'Souza, D. F.; Wills, A. C.: *Objects, Components and Frameworks with UML: The Catalysis Approach*. Addison-Wesley, 1998.
- [6] Tarr, P.; Ossher, H.; Harrison, W.; Sutton, S. M. Jr.: *N Degrees of Separation: Multi-Dimensional Separation of Concerns*. Proceedings of the 21st International Conference of Software Engineering, ICSE, Los Angeles, CA, USA, May 16-22, 1999. ACM 1999, pp. 107 – 119.
- [7] Andersen, E. P.; Reenskaug, T.: *System Design by Composing Structures of Interacting Objects*. Proceedings of the 6th European Conference on Object-Oriented Programming, ECOOP, Utrecht, The Netherlands, June 29 - July 3, 1992. LNCS Vol. **615**, Springer-Verlag 1992, pp. 133 – 152.
- [8] Harrison, W.; Ossher, H.: *Subject-oriented programming (a critique of pure objects)*. Proceedings of the 8th Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA, Washington, DC, USA, September 26 - October 1, 1993. SIGPLAN Notices **28**(10), 1993, pp. 411 – 428.
- [9] Nuseibeh, B.; Kramer, J.; Finkelstein, A.: *A Framework for Expressing the Relationships Between Multiple Views in Requirements Specifications*. In IEEE Transactions on Software Engineering, **20**(10), October 1994, pp. 760 – 773.
- [10] Lieberherr, K.: *Adaptive Object-Oriented Software. The Demeter Method with Propagation Patterns*. PWS Publishing Company, 1996.
- [11] Kiczales, G.; Lamping, J.; Mendhekar, A.; Maeda, C.; Lopes, C. V.; Loingtier, J.-M.; Irwin, J.: *Aspect-Oriented Programming*. Proceedings of the 11th European Conference on Object-Oriented Programming, ECOOP, Jyväskylä, Finland, June 9-13, 1997. LNCS Vol. **1241**, Springer-Verlag, 1997, pp. 220 – 242.
- [12] Mezini, M.; Lieberherr, K.: *Adaptive Plug-and-Play Components for Evolutionary Software Development*. Proceedings of the 13th Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA, Vancouver, British Columbia, Canada, October 18-22, 1998. SIGPLAN Notices **33**(10), 1998, pp. 97 – 116.
- [13] Ossher, H.; Tarr, P.: *Hyper/J: Multi-Dimensional Separation of Concerns for Java*. Proceedings of the 22nd International Conference on Software Engineering, ICSE, Limerick Ireland, June 4-11, 2000. ACM 2000, pp. 734 – 737.
- [14] Kiczales, G.; Hilsdale, E.; Hugunin, J.; Kersten, M.; Palm, J.; Griswold, W. G.: *An Overview of AspectJ*. Proceedings of the 15th European Conference on Object-Oriented Programming, ECOOP, Budapest, Hungary, June 18-22, 2001. LNCS Vol. **2072**, Springer-Verlag, 2001, pp. 327 – 353.
- [15] Miller, J.; Mukerji, J.: *Model Driven Architecture (MDA)*. Object Management Group, Draft Specification ormsc/2001-07-01, July 9, 2001.
- [16] Object Management Group, Inc.: *Model Driven Architecture*. <http://www.omg.org/mda>.
- [17] Sendall, S.; Strohmeier, A.: *UML-based Fusion Analysis*. Proceedings of the 2nd International Conference on The Unified Modeling Language: Beyond the Standard, UML, Fort Collins, CO, USA, October 28-30, 1999. LNCS Vol. **1723**, Springer-Verlag, 1999, pp. 278 – 291. An extended version is also available as Technical Report, EPFL-DI No 99/319.
- [18] Sendall, S.; Strohmeier, A.: *Fondue*. <http://lgl.epfl.ch/research/fondue/index.html>.
- [19] Coleman, D.; Arnold, P.; Bodoiff, S.; Dollin, C.; Gilchrist, H.; Hayes, F.; Jeremaes, P.: *Object-Oriented Development: The Fusion Method*. Prentice Hall, 1993.
- [20] Object Management Group, Inc.: *Unified Modeling Language Specification*, v1.4, September 2001.
- [21] Warmer, J.; Kleppe, A.: *The Object Constraint Language: Precise Modeling with UML*. Addison-Wesley, 1998.
- [22] Rumbaugh, J.; Blaha, M.; Premerlani, W.; Eddy, F.; Lorenson, W.: *Object-Oriented Modeling and Design*. Prentice Hall, 1990.
- [23] Kruchten, P. B.: *The Rational Unified Process: An Introduction*. Addison-Wesley, 1998.
- [24] Jacobson, I.; Booch, G.; Rumbaugh, J.: *The Unified Software Development Process*. Addison-Wesley, 1999.
- [25] Jacobson, I.; Christerson, M.; Jonsson, P.; Övergaard, G.: *Object-Oriented Software Engineering: A Use Case Driven Approach*. Addison-Wesley, 1992.
- [26] Booch, G.: *Object-Oriented Analysis and Design with Applications (2nd edition)*. Addison-Wesley, 1994.
- [27] Kruchten, P. B.: *The 4+1 View Model of Architecture*. IEEE Software, **12**(6), 1995, pp. 42 – 50.
- [28] Beck, K.: *Extreme Programming Explained: Embrace Change*. Addison-Wesley, 1999.
- [29] Atkinson, C.; Bayer, J.; Laitenberger, O.; Zettel, J.: *Component-Based Software Engineering: The Kobra Approach*. Workshop on Component-Based Software Engineering, at the International Conference on Software Engineering, Limerick, Ireland, June 4-11, 2000. <http://www.sei.cmu.edu/cbs/cbse2000/>.
- [30] Atkinson, C.; Bayer, J.; Bunse, C.; Kamsties, E.; Laitenberger, O.; Laqua, R.; Muthig, D.; Paech, B.; Wust, J.; Zettel, J.: *Component-based Product-line Engineering with UML*. Addison-Wesley, 2001.
- [31] Deck, M.: *Cleanroom and Object-Oriented Software Engineering: A Unique Synergy*. Proceedings of the 8th Annual Software Technology Conference, Salt Lake City, UT, USA, April 1996.
- [32] Graham, I.; Henderson-Sellers, B.; Younessi, H.: *The OPEN Process Specification*. Addison-Wesley, 1997.
- [33] Bunse, C.; Atkinson, C.: *Improving Quality in Object-Oriented Software: Systematic Refinement and Translation of Models to Code*. Proceedings of the 12th International Conference on Software & Systems Engineering and their Applications, ICSSEA, Paris, France, December 1999.

- [34] Bunse, C.: *Pattern-Based Refinement and Translation of Object-Oriented Models to Code*. PhD Thesis, Fraunhofer IRB-Verlag, 2001.
- [35] Stojanovic, Z.; Dahanayake, A.; Sol, H.: *Integration of Component-Based Development Concepts and RM-ODP Viewpoints*. Proceedings of the 1st International Workshop on Open Distributed Processing: Enterprise, Computation, Knowledge, Engineering and Realisation, WOODPECKER, at the International Conference on Enterprise Information Systems, ICEIS, Setúbal, Portugal, July 7-10, 2001. ICEIS Press 2001, pp. 98 — 109.
- [36] Rouvellou, I.; Sutton, S. M. Jr.; Tai, S.: *Multidimensional Separation of Concerns in Middleware*. Workshop on Multidimensional Separation of Concerns in Software Engineering, at the International Conference on Software Engineering, Limerick, Ireland, June 4-11, 2000. <http://www.research.ibm.com/hyperspace/workshops/icse2000/>.
- [37] Kovse, J.: *Generic Model-to-Model Transformations in MDA: Why and How?*. Workshop on Generative Techniques in the context of Model-Driven Architecture, at the Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA, Seattle, WA, USA, November 4-8, 2002. <http://www.soft-metaware.com/oopsla2002/mda-workshop.html>.
- [38] Silaghi, R.: *Generic Concern-Oriented Model Transformations Meet AOP*. Proceedings of the 1st International Workshop on Model-driven Approaches to Middleware Applications Development, MAMAD, at the International Middleware Conference, Rio de Janeiro, Brazil, June 16-20, 2003. PUC-Rio Press 2003, Middleware 2003 Companion, pp. 307 — 311.
- [39] First International Workshop on *Aspect-Oriented Modeling with UML*, in conjunction with the 1st International Conference on Aspect-Oriented Software Development, Enschede, The Netherlands, April 22-26, 2002. <http://lglwww.epfl.ch/workshops/aosd-uml/index.html>.
- [40] Second International Workshop on *Aspect-Oriented Modeling with UML*, in conjunction with the 5th International Conference on the Unified Modeling Language - the Language and its Applications, Dresden, Germany, September 30 - October 4, 2002. <http://lglwww.epfl.ch/workshops/uml2002/index.html>
- [41] Rational Software Corporation: *UML Profile for EJB*, Public Draft, May 2001.
- [42] Object Management Group: *UML Profile for CORBA Specification*, v1.0, October 2000.
- [43] Kienzle, J.; Guerraoui, R.: *AOP: Does it Make Sense? The Case of Concurrency and Failures*. Proceedings of the 16th European Conference on Object-Oriented Programming, ECOOP, University of Málaga, Spain, June 10-14, 2002. LNCS Vol. **2374**, Springer-Verlag, 2002, pp. 37 — 61.